(2)

Task/Subtask ID52.1(2)
CDRL Sequence 05504-001
31 July 1993

SOFTWARE TECHNOLOGY FOR ADAPTABLE, RELIABLE SYSTEMS (STARS) PROGRAM

Cleanroom Engineering Handbook
Volume 5
Development Team Practices

Contract No. F19628-88-D-0032

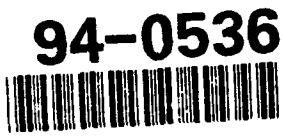
Task ID52 – STARS Technology Transfer Demonstration Project for the U.S. Army

Prepared for:

Electronic Systems Center
Air Force Materiel Command, USAF
Hanscom AFB, MA 01731-2816

Prepared by:

IBM Federal Systems Company
800 North Frederick Avenue
Gaithersburg, MD 20879



Task/Subtask ID52.1(2)
CDRL Sequence 05504–001
31 July 1993

SOFTWARE TECHNOLOGY FOR ADAPTABLE, RELIABLE SYSTEMS (STARS) PROGRAM

Cleanroom Engineering Handbook
Volume 5
Development Team Practices

Contract No. F19628–88–D–0032

Task ID52 – STARS Technology Transfer Demonstration Project for the U.S. Army

Prepared for:

Electronic Systems Center
Air Force Materiel Command, USAF
Hanscom AFB, MA 01731–2816

Prepared by:

IBM Federal Systems Company
800 North Frederick Avenue
Gaithersburg, MD 20879

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0166

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	7/31/93	Initial

4. TITLE AND SUBTITLE
Cleanroom Engineering Handbook:
Development Team Practices

5. FUNDING NUMBERS
F19628-88-C-0032/0010

6. AUTHOR(S)
Ara Kouchakdjian
Richard H. Cobb
Alan R. Hevner
James A. Whittaker

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
IBM Federal Systems Company
800 North Frederick Avenue
Gaithersburg, MD 20879

SET, Inc.
2770 Indian River Blvd.
Vero Beach, FL 32960

8. PERFORMING ORGANIZATION REPORT NUMBER
05504-001
Volume 5

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Electronic Systems Center/ENS
Air Force Materiel Command, USAF
5 Eglin Street, Building 1704
Hanscom Air Force Base, MA 01731-2116

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES
N/A

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Cleared for Public Release, Distribution is Unlimited

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

This is one of a series of six engineering handbooks prepared for and used by the engineering staff at Picatinny Arsenal for the STARS technology transfer demonstration. The handbooks define the engineering process and algorithms that will be used in Cleanroom projects. They are designed to provide support to trained engineers using Cleanroom Engineering, not to substitute for training.

This handbook, Volume 5, explains the set of specific tasks performed by the Cleanroom Development Team to design and implement each increment j in the software project. In the Cleanroom environment, the Development Team has a well-defined mission which can be stated as: "Given a set of functions (i.e., specifications) which are to be implemented in software, find rules (i.e., program code) that correctly implement the functions".

State-of-the-art systems engineering and software engineering principles, methods, and tools are employed in the Cleanroom development process. The theory and methods of box structure design objects that the development team utilizes are defined. Templates for preparing all design tasks are defined. Tasks for correcting software failures during certification are described.

The Cleanroom process model for software system development projects is presented in Volume 1 - Cleanroom Process Overview - of this series of handbooks. This handbook, Volume 5, describes the activities of Cleanroom software development for increment j. Process P.5.j, Software Development and Certification Development for Increment j. provides the framework for transforming system and software specifications (described in Volume 4 - Specification Team Practices) into design objects and code implementation.

14. SUBJECT TERMS	15. NUMBER OF PAGES
Certification, Cleanroom, Cleanroom Engineering, Development, Management, Software Development, Specification	71
	16. PRICE CODE: N/A

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	SAR

PREFACE

This series of handbooks is prepared for use by managers and engineers assigned to Cleanroom projects at Picatinny Life Cycle Software Engineering Center.

These handbooks define the engineering process and algorithms that will be used in Cleanroom projects.

This document was developed by the IBM Federal Systems Company, located at 800 North Frederick Avenue, Gaithersburg, MD 20879 and Software Engineering Technology, Inc. located at 2770 Indian River Boulevard, Vero Beach, FL 32960. Questions or comments should be directed to Mr. Paul Arnold at 301-240-7464 (Internet: pga@sei.cmu.edu).

DEVELOPMENT TEAM PRACTICES

TABLE OF CONTENTS

DEVELOPMENT TEAM PRACTICES

SECTION 1: INTRODUCTION

The mission of this handbook for Cleanroom Development is to organize and explain the set of specific tasks performed by the Cleanroom Development Team to design and implement each increment j in the software project.

In the Cleanroom environment, the Development Team has a well-defined mission which can be stated as:

> Given a set of functions (i.e., specifications) which are to be implemented in software, find rules (i.e., program code) that correctly implement the functions.

State-of-the-art systems engineering and software engineering principles, methods, and tools are employed in the Cleanroom development process. The theory and methods of box structure design objects that the development team utilizes are defined. Templates for preparing all design tasks are defined. Tasks for correcting software failures during certification are described.

The Cleanroom process model for software system development projects is presented in *Volume 1 - Cleanroom Process Overview* of this series of handbooks. This volume 5 describes the activities of Cleanroom software development for increment j. Process P.5.j, Software Development and Certification Development for Increment j, provides the framework for transforming system and software specifications (described in *Volume 4 - Cleanroom Software Specification*) into design objects and code implementation.

1.1 Box Structures: Some Background

People have been searching for methods that they can use to guide the development of software solutions. Today most competent software developers are using some variant of one of the following three methods:

Process Oriented Development
- Example methods include:
  - Structured Analysis and Structured Design
  - Jackson System Development

Data Oriented Development
- Example methods include:
  - Warnier-Orr Systems Development
  - Information Engineering

Object Oriented Development
- Example methods include:

Booch's Design Method
Meyer's Approach (Eiffel)
Coad and Yourdan Methods

Some practitioners using these methods design good software solutions and others are not so successful. The difficulty is that the software development problem is multifaceted and each of these three orientations focus on just one facet giving only limited emphasis on the other two facets.

Mills developed the Box Structure Method to unify the three approaches by providing the engineer with the ability to focus on each view in turn. The relationship between the Box Structure views and the traditional software design methods is as follows:

Black Box	Object orientation
State Box	Data orientation
Clear Box	Process orientation

Box structures support a rigorous, yet practical, set of methods for the development of systems. Box structure methods have been used successfully on numerous projects.

The box structure design algorithm, as presented in this manual, combined with all other Cleanroom practices permit software engineers to replace the craft-based activities that they currently use to develop object-oriented systems with engineering-based processes. As a result they derive all the benefits that engineering provides over crafting.

1.2 Cleanroom Development Process Model - P5.j

The Cleanroom development process model is embedded within the P5.j process, Software Development and Certification Development for Increment j.

proc P5.j: Software Development and Certification Preparation
[For each P5.j, the specification is tailored, then the software is designed to the code by the Development team (P5.j.3), while the Certification team does the work necessary to prepare for certification of the increment (P5.j.2).]
do [P5.j: Software Development and Certification Preparation]
run P5.j.1: Tailor specification to increment/accumulation j;
con
run P5.j.2: Prepare for Certification of Accumulation j;
run P5.j.3: Increment j Development;
noc;
until
Completion Conditions achieved for P5.j.2 and P5.j.3 **or** P5.j.3 team decision indicates the need for replanning or specification revisions
od;
corp;

A brief summary of the three subprocesses is presented. The next section then describes P5.j.3, Increment j Development, in detail.

1.2.1 Tailor Specification to Increment j

The Specification Team determines the set of increments for the development of the complete system. The description of system increments is contained in *Volume VI - Construction Plan* of the system specification. As each increment is defined the Specification Team will tailor the specification information for that increment into an effective format for the Development Team and the Certification Team to use in their processes. The model for this subprocess is:

proc P5.j.1: Tailor specification to increment/accumulation j
do [P5.j.1: Tailor specification to increment/accumulation j]
con
S5.j.1.1: Tailor Black Box functions to increment/accumulation j;
S5.j.1.2: Tailor Usage Profile to increment/accumulation j;
noc;
until
Completion Conditions achieved for S5.j.1.1 and S5.j.1.2
od;
corp;

A full description and explanation of these tasks are found in the process manual *Volume 4 - Cleanroom Software Specification.*

1.2.2 Prepare for Certification of Accumulation j

In parallel with increment j development, the Certification Team will prepare for the certification of the accumulation of increments 1,...,j by constructing the test plan and test scenarios. The process model for this subprocess is:

```
proc P5.j.2: Prepare for Certification of Accumulation j
   [This process results in test plan and test scenarios for accumulation j]
   [P5.j.2:  Prepare for Certification of Accumulation j]
   con
      do
         do
            C5.j.2.1:    Prepare test plan;
         until
            Completion Conditions for C5.j.2.1 achieved
         od;
         do
            con
               C5.j.2.2:    Prepare test scenarios or test case generator for accumulation j;
               C5.j.2.3:    Determine expected results for test cases;
            noc;
         until
            Completion Conditions for C5.j.2.2 and C5.j.2.3 achieved
         od;
      od;
      C5.j.2.4:    Increase Understanding of Problem and Solution Domains;
   noc;
corp;
```

A full description and explanation of these tasks are found in the process manual *Volume 6 - Cleanroom Software Certification.*

1.2.3 Increment j Development

This subprocess contains the tasks that are the focus of this manual. The process model for increment j development is:

proc P5.j.3: Increment j Development
[Process results in verified software for increment.]
do [P5.j.3: Increment Development]
con
for all design objects currently available to be worked on
do
D5.j.3.1: Select a box design object from pick list; [Items appear on individual developers pick list based on (1) Completion Conditions achieved for parent design object, (2) design object previously assigned to team member making selection and (3) design object is not yet fully refined]
D5.j.3.2: **run** P5.j.3.2 Refine and verify box;
D5.j.3.3: Update pick list; [To reflect results of D5.j.3.2]
od;
D5.j.3.4: Increase Understanding of Problem and Solution Domains;
D5.j.3.5: Team Decision if one is required: (1) Increment complete, verified and ready for certification, (2) Design problems-project/spiral should be replanned or (3) Specification problems-specifications should be revised;
noc;
until
D5.j.3.5 team decision indicates increment complete **or** project should be replanned **or** specifications should be revised
od;
corp

Sections 2 and 3 will describe the above five tasks in detail. The D prefix indicates that the Development Team is responsible for the performance of the task.

1.2.4 Verifying Failures Found During Certification of Increment j

This task is also discussed in this manual.

D6.j.4: Correct failure, verify correction and prepare ECN;

This task is a part of the P6 process. It will be discussed in section 5.

SECTION 2: INCREMENT J DEVELOPMENT TASKS

The development process followed by the Cleanroom Development Team is contained in process P5.j.3, Increment j Development. The detailed process is repeated here:

proc P5.j.3: Increment j Development
[Process results in verified software for increment.]
do [P5.j.3: Increment Development]
con
for all design objects currently available to be worked on
do
D5.j.3.1: Select a box design object from pick list; [Items appear on individual developers pick list based on (1) Completion Conditions achieved for parent design object, (2) design object previously assigned to team member making selection and (3) design object is not yet fully refined]
D5.j.3.2: **run** P5.j.3.2 Refine and verify box;
D5.j.3.3: Update pick list; [To reflect results of D5.j.3.2]
od;
D5.j.3.4: Increase Understanding of Problem and Solution Domains;
D5.j.3.5: Team Decision if one is required: (1) Increment complete, verified and ready for certification, (2) Design problems-project/spiral should be replanned or (3) Specification problems-specifications should be revised;
noc;
until
D5.j.3.5 team decision indicates increment complete **or** project should be replanned **or** specifications should be revised
od;
corp

The five development tasks that are identified in the process are discussed in this section. The fundamental box structure concepts, methods, and principles that underlie these tasks are reviewed in Sections 3 and 4 of this manual.

2.1 Select Box Design Object from Pick List

The Cleanroom Development Team builds the increment design in a top-down manner in the framework of a box structure usage hierarchy. The critical principle of *referential transparency* allows the team manager to assign the development of box structures as independent units of design. This provides clean lines of delegated responsibility for units of work. Thus, each unique box structure in the increment's usage hierarchy is an assignable design object.

An engineer on the development team is assigned one or more box structures for analysis and design. The engineer commences work by selecting an available box structure design object from his or her pick list. This selection process is based on priority assignment by the team manager or by selecting the box structure that is highest in the hierarchy and still uncompleted.

2.2 Refine and Verify Box

The process for refining and verifying each box structure in the system design is:

proc P5.j.3.2: Refine and Verify Box
[Process results in a refined and verified box.]
do [P5.j.3.2: Refine and Verify Box
D5.j.3.2.1: Refine, verify and team review design object;
if team review passes
then
D5.j.3.2.2: Sign Completion Conditions;
else
D5.j.3.2.3: Team Decision: (1) Continue refinement of design object, (2) Change some prior design decision which requires pruning of design hierarchy;
fi;
until
Completion Conditions signed by full team **or** team review concludes that a prior design decision be modified which requires pruning of design hierarchy and updating of pick list
od;
corp;

Each of these sub-tasks are described in the following sections.

2.2.1 Refine Design Object

The analysis, design, verification, and implementation of a box structure object progresses through a series of tasks. These activities are detailed in the *Box Structure Algorithm*:

Box Structure Algorithm

I. Develop Black Box
con
Task 1 - Define Stimuli and Responses
Task 2 - Develop Black Box Behavior
Task 3 - Examine and Verify Black Box
noc;

II. Develop State Box

con

Task 4 - Determine State Data
Task 5 - Retain/Migrate State Data
Task 6 - Develop State Box Behavior
Task 7 - Examine and Verify State Box

noc;

III. Develop Clear Box

con

Task 8 - Identify Data Objects and Internal Black Boxes
Task 9 - Develop Clear Box Behavior
Task 10 - Examine and Verify Clear Box

noc;

IV. Refine Clear Box

con

Task 11 - Refine Clear Box
Task 12 - Examine and Verify Refined Clear Box

noc;

Repeat Step IV Until Clear Box Completed

V. Box Structure Translation

con

Task 13 - Translate Design
Task 14 - Verify Translation

noc;

Repeat Algorithm for Each Internal Black Box

End of Box Structure Algorithm

The work involved with each of these steps is described in Section 4 of this manual. The Cleanroom engineer is responsible for executing the steps of the algorithm on the assigned box structure object. Each step further refines the object to a final design for software implementation.

The engineer retrieves and returns the design object to the central repository in the Cleanroom development environment. The current status of the object (i.e., the current work step) is maintained in the system along with all intermediate work products.

2.2.2 Verify Refinement

At well-defined points in the box structure refinement, the design object must be verified for correctness and consistency. These points are:

Task 3 - Examine and Verify Black Box
Task 7 - Examine and Verify State Box
Task 10 - Examine and Verify Clear Box
Task 13 - Examine and Verify Refined Clear Box
Task 15 - Examine and Verify Translation

Each of these steps is critical and must be completed before the box structure development can continue. Section 4 provides guidance for presenting effective verification arguments. These arguments are important for the team proof reviews that evaluate Completion Conditions for each box structure object.

2.2.3 Box Design Language

A specific language has been created to document desired behavior using box structures. The language is called Box Design Language (BDL). Its syntax is closely related to other third generation languages, so should appear quite intuitive. The BNF for BDL appears in Appendix A.

2.2.4 Team Proof Reviews

A *Team Proof Review* is performed for each box structure design object in the increment. The complete Cleanroom development team, or a selected portion thereof, meets to evaluate and determine the completion of the design object. The box structure is evaluated in terms of the *four products of the development process*: the black box, the state box, the (refined) clear box, and the software implementation (e.g., code).

A team review is a simple activity. The team assembles for the meeting. It is not required that the material to be reviewed be distributed before the review, although in some cases that may be a good thing to do. The engineer whose material is to be reviewed presents the material with the goal of confirming to the rest of the team that the design is correct. If the team is persuaded, then the review passes. Otherwise, the engineer will need to modify material and have it reviewed again. Reviews should be timed so they do not go on for more than one hour. The amount of work to be reviewed should fit into that time period, which also drives the frequency at which reviews will occur.

The reasons that a review does not pass can include the following: a mistake has been found, a design is so complex that the team could not be persuaded of its correctness, or the review has taken too much time. Actions to be taken for a change are discussed in Section 2.2.5. If a review passes, then Completion Conditions can be signed, as described in Section 2.2.4.

The Completion Conditions are determined to be satisfied, as discussed in Section 2.2.4, or changes are required in the box structure design, as discussed in Section 2.2.5.

2.2.5 Sign Completion Conditions

If the team review of the design object is successful, a document is signed stating that all Completion Conditions are met. All team members that participated in the review must agree that all conditions are satisfied. The four development products (i.e., black box, state box, refined clear box, and implementation) are stored in the development repository. The pick list is updated to reflect the completion of the current design object. This may release additional box structure objects (e.g., the internal black boxes of the current object) for development.

The following describes typical Completion Conditions that must be satisfied for each product to be completed.

Black Box Completion Conditions

1. Is the black box presented in appropriate box description language (BDL)?

2. Are all stimuli and responses identified, clearly labeled with meaningful names, and fully described?

3. Have all black box behaviors been clearly defined in terms of stimulus histories?

4. Have sufficient black box analyses been performed? Has black box closure been verified?

5. Are the black box verification arguments complete, accurate, and clear?

State Box Completion Conditions

1. Is the state box presented in appropriate box description language (BDL)?

2. Are all required state data designed, clearly labeled with meaningful names, and fully described?

3. Are the design decisions on state data grouping good ones? Is the set of state data to be maintained in the current box structure cohesive and meaningful?

4. Have all state box behaviors been clearly defined in terms of current stimulus and current state?

5. Have sufficient state box analyses been performed? Has state box closure been verified? Has a state usage analysis been performed?

6. Does the state box process all stimuli values, valid and invalid, leaving the state data in a valid condition?

7. Are the state box verification arguments complete, accurate, and clear?

Clear Box Completion Conditions (These conditions must also be satisfied for each refinement of the clear box.)

1. Is the clear box presented in appropriate box description language (BDL)?

2. Are all procedural constructs in the clear box clearly described?

3. Are the design decisions on the internal black boxes good ones? Do the internal black boxes define cohesive, independent behaviors? Do the internal black boxes support effective state migration?

4. Have all clear box behaviors been clearly defined in terms of current stimulus and current state?

5. Have sufficient clear box analyses been performed? Has clear box closure been verified? Is referential transparency clearly supported in the clear box?

6. Have all object composition opportunities and common service opportunities been explored?

7. Has the use of concurrent behaviors in the clear box been exploited? If so, are appropriate concurrency controls in place?

8. Are the clear box verification arguments complete, accurate, and clear?

Implementation Completion Conditions

1. Is the implementation in an appropriate structured programming language?

2. Is the code presented in a readable form with structured indentation and adequate comments?

3. Is the code modularized for easy understanding and maintenance?

4. Are the code verification arguments complete, accurate, and clear?

2.2.6 Team Decision Options for Design Object

If the team review of the design object is not completely successful, a decision is made on what needs to be done to complete the object. Two options are possible:

Option 1: The development team should continue the design process with additional refinements to the box structure products. The review team should identify clearly the problems found and the steps needed to complete the object development.

Option 2: The review team may discover problems in the increment design that are not solely isolated in the current design object. Design decisions made in higher levels of the box structure usage hierarchy may be called into question. The impact of the needed changes should be studied along with the identification of all design objects that must be modified. The objects to be modified are placed on the pick list along with specifications for the needed modifications.

2.3 Update Pick List

Based on the decisions made in the previous task, the development team pick list is updated. New design objects (e.g., the newly defined internal black boxes) are released for development or existing design objects are placed on the pick list for further refinement.

2.4 Increase Understanding of Problem and Solution Domains

Many issues and questions arise during all phases of the Cleanroom development process. These issues are often the result of some new, unknown factor being noticed. This factor may become a part of the design or it may not. But to make any sort of determination, the issue must be understood. As a result, the development team must find clarifications and answers expeditiously.

The first task required of the development team when faced with a issue or question is to fully understand it. This will require talking with the individual(s) who raised the problem. If it is a simple misunderstanding, the matter can be settled by a clarification and complete answer.

Deeper system issues will necessitate an increased understanding of the problem domain and the solution domain. A development team member, or the entire development team may be required to go off and perform an analysis task. This will typically include information through interviews, document reviews, and prototyping. Also, more information on solution opportunities can be gathered. This new information is integrated with the known information to create an analysis report that documents what has been learned. The task of increasing the understanding of system requirements continues until the resolutions to the questions are considered satisfactory.

2.5 Team Decision Options for Increment j

The development tasks continue as described in the *Increment j Development* process until one of three conditions occur:

Option 1: All design objects in Increment j are completed and signed-off in team reviews. The increment is the passed to the Cleanroom Certification Team for testing and certification.

Option 2: New understandings of the problem and solution domains require significant re-specification of Increment j. All intermediate design products are maintained in their current state for possible use when Increment j comes under development again. The Specification Team is called on to develop modified specification for increment j.

Option 3: Project planning issues (e.g., budget, personnel, materials) require that the increment development be re-planned. Re-planning tasks are covered in *Volume 2* of the Process Manuals. Again, all intermediate design products are maintained in their current state for possible use when Increment j comes under development again.

Section 3: Box Structure Overview

Box-structured systems development is a stepwise refinement and verification process that produces a system design. Such a system design is defined by a hierarchy of small design steps that permit the immediate verification of their correctness. Three basic principles underlie the box-structured design process:

1. All data to be defined and stored in the design are hidden in data abstractions.

2. All processing is defined by sequential and concurrent uses of data abstractions.

3. Each use of a data abstraction in the system occupies a distinct place in the usage hierarchy of the system.

These principles are embodied in box structure system views and methods for constructing box structures.

3.1 Box Structure Concepts

Box structure methods define a single data abstraction in three forms in order to isolate the creative design steps involved in building the abstraction. The **black box** gives an external description of data abstraction behavior in terms of a mathematical function from stimulus histories to responses. The black box is the most abstract description of system behavior and can be considered as a requirements statement for the (sub)system. The **state box** includes a designed state and an internal black box that transforms the stimulus and an initial state into the response and a new state. The state is designed from an analysis of the required stimulus histories and responses for the system. Finally, the **clear box** replaces the internal black box with the designed sequential or concurrent usage of other black boxes as subsystems. These new black boxes are expanded at the next level of the system **box structure usage hierarchy** into state box and clear box forms.

Box structures have underlying mathematical foundations that permit the scale-up of analysis and design to systems of arbitrary size. These foundations are based on sets and functions that can be described in mathematical notation for small systems or subsystems or in well-structured natural language in a given context in larger systems.

Throughout the box structure design process, it is important to preserve design information and knowledge as the design is refined from black box to state box to the initial clear box and all clear box refinements. This is done by retaining the black box and state box statements as *intended functions*. These design statements are included as comments at appropriate points throughout the BDL.

Intended functions basically serves to describe portions of a design in a more abstract/informal (but still precise form). There are two places where intended functions appear, before and after a line or structure of the box structure they are describing. The more abstract/informal form that appears before a line or structure is typically the design from a previous step. Intended functions may also appear after a line or structure to provide rigor for a portion of the design that is presently not rigorously enough described. Basically, this type of intended function serves as a surrogate for stimulus history or other information in line. For example, a condition in terms of stimulus history may need a comment that will more rigorously define the stimulus history. This type of intended function is underlined, so a reader will be aware that it is necessary to understand the design statement appearing before it.

Providing intended functions result in a precise description of behavior. Typical function commentary are non-rigorous editorials that attempt to describe behavior. A precise, yet readable, description of behavior will provide the stakeholders in that design with useful information, not an editorial. Examples of intended functions appear below:

Intended Function:

B08.06 **then** [Update and return files]
B08.07 **if** Version# = Version# of S7 stimulus in B8.1
B08.08 **then** [version# is same as most recent version]

Intended Function:

B09.09 RA2: Proj_Sys_Com(copy, SDR_Directory/Projid/Path_info, File_name||Version#, S_Directory, File_name||Version#);
[where S_directory is (Workstation_Machine_Name||Session_Directory) from most recent S1->R1(000 'Project Session opened')]
[where Version# is requested Version# or latest Version# if Version# empty.]
[where Projid is from most recent S1->R1(000)]

3.2 Box Structure Principles

The effective use of box structures for the development of information systems is guided by the use of nine basic box structure principles, referential transparency, transaction closure, state migration, common services, correct design trail, functional verification, writing correct programs - not proving programs correct, proof by direct assertion, reorganization. The principles are:

Referential Transparency - Referential transparency occurs when a black box abstraction is completely defined within the clear box at the next higher level in the usage hierarchy. The black box is then logically independent of the rest of the system, and can be designed to satisfy a well defined behavior specification. The principle of referential transparency provides a crisp discipline for management delegation and assignment of responsibility.

Transaction Closure - The principle of transaction closure defines a systematic, iterative specification process to ensure that a sound and complete set of transactions is identified to achieve the required system behavior. The closure process can be performed at each box structure view of an object abstraction. At the black box, checks are performed to ensure that the system stimuli are necessary and sufficient to generate the required system responses. At the state box, the defined transactions must be necessary and sufficient for the acquisition and preservation of all state data, and the state data must be necessary and sufficient for the completion of all transactions. At the clear box, the procedural design and the internal black boxes must include all transactions.

State Migration - State data should be identified and stored in the system part (i.e., data abstraction) at the lowest level in the box structure hierarchy that includes all references to that data. At any time in the systems development process, state data can be migrated upward or downward in the hierarchy in order to achieve some system objective, such as minimizing data scope. State migration must be performed carefully in order to maintain the consistency and mathematical correctness of data abstractions throughout the hierarchy.

Common Services - A common service is a data abstraction that is described in a separate box structure hierarchy, and used in other box-structured systems. System parts with multiple uses should be defined as common services for reusability. Also, predefined common services, such as database management systems and input/output interfaces, should be used to advantage throughout the box structured system. The advantages of reusable common services for systems development are obvious. Box structures directly support the identification and reuse of common services within and among systems.

Correct Design Trail - When developing software while using Cleanroom, just like when using any other approach, errors can occur. When correcting an error, it is important to ensure that the entire design trail continues to be kept consistent. For example, if while developing a state box, one discovers that a stimulus is missing, then all previous documents, from the first onwards, must be corrected. If corrections to a previous design document are trivial, then they can be made in real time, without stopping the current design process. If the corrections are major, then the current design activity stops, until the previous documents are made correct.

Functional Verification - A program is a rule for a mathematical function. Looking at any part of a design or program allows the power of mathematics to confirm correctness. An additional benefit of looking at any part of a design/program as a rule, is that it defines a specified behavior. As a result of referential transparency, we determine that a program's behavior stays unchanged whether or not it is changed. That means that we need only to confirm that a change (such as a refinement) was made correctly. Therefore, the amount of verification is only a function of the changes made between refinements, and not a function of the size of the design or program.

Writing Correct Programs, Not Proving Programs Correct - In the subsequent sections, one will notice that the design tasks are kept separate from the analysis and verification tasks. The reason

for this is a documentation separation issue, not necessarily a task separation issue. It is critical for an engineer to do all of the tasks for a particular step in concert. In that manner, the design is created consistently with the analysis and verification. Complete separation of these tasks results in a design being written, followed by the activity of proving the program correct. Writing the program with the proof in mind, increases the probability of creating a correct design, and facilitating its proof, often by direct assertion.

Proof by Direct Assertion - Any structured program can be verified, as proven in Structured Programming by Linger, Mills and Witt. But, functional verification is not a trivial task, and, because it is rigorous, can be time-consuming. The goal in developing software is to write code that will be verifiable by direct assertion, and not by written proof. This is done by taking small design steps, rather than large leaps, and by creating designs with the understanding that both the developer and other members of the development team will need to confirm that the refinement was correct. Creating simple and easy to read designs are easy to verify, often by direct assertion, and are typically correct.

Box Structure Reorganization - What will appear in the sections below are strategies for box structure refinements. It is also possible that a box can be reorganized. What this means is that some part of the control structure may be modified, but all modifications do not change the behavior of the box. For example, the first black box for the development team is prepared by the specification team. It may not be in a form most conducive for the development team to use. The development team can modify the black box to make it more conducive for development. Analyzing and verifying the correctness of a reorganization requires the same tasks as analysis and verification of a refinement (that is, black to clear box, state to black box, clear to state box, or clear box refinement to clear box). All descriptions of design refinements can also be viewed as reorganizations. More specifically, creating the state box from the black box entails the same process as creating the state box from another state box.

Quality and Correctness - Hand in hand with the inventive steps, developers also confirm that the invention was acceptable. The confirmation is done by examination and verification. Verification confirms that the invention was correct. Examination confirms that the invention was the best possible. Having an appraisal from both of these perspectives ensures that the invention was correct and the best possible.

These principles will be referenced throughout the box structure design steps found in the next section.

Section 4: Box Structure Software Development

The *Box Structure Algorithm* is repeated here with an index to the section and page where each step is described. One can view it as a five step algorithm, with a total of 15 tasks.

Box Structure Algorithm

I. Develop Black Box
con
Task 1 - Define Stimuli and Responses - Sect. 4.1.1, p. 21
Task 2 - Develop Black Box Behavior - Sect. 4.1.2, p. 23
Task 3 - Examine and Verify Black Box - Sect. 4.1.3, p. 25
noc;

II. Develop State Box
con
Task 4 - Determine State Data - Sect. 4.2.1, p. 27
Task 5 - Retain/Migrate State Data - Sect. 4.2.2, p. 28
Task 6 - Develop State Box Behavior - Sect. 4.2.3, p. 29
Task 7 - Examine and Verify State Box - Sect. 4.2.4, p. 31
noc;

III. Develop Clear Box
con
Task 8 - Identify Data Objects and Internal Black Boxes - Sect. 4.3.1, p. 36
Task 9 - Develop Clear Box Behavior - Sect. 4.3.2, p. 38
Task 10 - Examine and Verify Clear Box - Sect. 4.3.3, p. 40
noc;

IV. Refine Clear Box
con
Task 11 - Refine Clear Box - Sect. 4.4.1, p. 45
Task 12 - Examine and Verify Refined Clear Box - Sect. 4.4.2, p. 47
noc;

Repeat Step IV until Clear Box Completed - Sect. 4.4.3, p. 50

V. Design Translation
con
Task 13 - Translate Design - Sect. 4.5.1, p. 50
Task 14 - Verify Translation - Sect. 4.5.2, p. 52
noc;

Repeat Box Structure Algorithm - Sect. 4.5.3, p. 53
for Each Internal Black Box

End of Box Structure Algorithm

In this section, a full description of each of the steps in the context of the tasks is provided along with templates for recording the results of box structure development.

4.1 Black Box Definition

A black box is defined by a mathematical function from histories of stimuli to the next response. Let S be the set of possible stimuli, and R be the set of possible responses of a system or subsystem. In illustration, an airlines reservation system with many thousands of concurrent users, will accept their stimuli sequentially into the system in real time and return responses accordingly. The black box function, say f, will map historical sequences of such stimuli, in this case S*, to responses, R, shown in the form

$$f: S^* \rightarrow R.$$

The description of function f may be very complex for many systems, e.g., an airlines reservation system, but it is a function, no more, no less. This description of the black box assumes no data storage between stimuli, even though such storage may be known to exist, or be planned for development.

The Cleanroom Specification Team performs specification tasks to understand and describe the requirements of the system and software to be developed. They produce a six-volume specification as described in *Volume 4* of the Cleanroom Process Manuals. The software specification is presented as black box behaviors that are tailored by the Specification Team for development and certification. Given a specification in terms of a high level black box requirement, or a clear box that defines the stimuli to lower level black boxes, the Development Team performs tasks to produce a black box design. The tasks are not necessarily completed in the order presented below, they are done in the order that most effectively results in the work getting done completely and correctly.

4.1.1 Task 1 - Define Stimuli and Responses

The black box of the system is completely defined based on the requirements for the system. The black box is described by its stimuli, responses, and the behaviors that map stimulus histories

into responses. The discovery of stimuli, responses, and behaviors is an iterative, interdependent process. For example, the identification of a new stimulus may cause a new response to be discovered, and vice versa.

Stimuli and responses are defined as variables. The following discussion describes an effective method for numbering variable names in box structure analysis and design.

Variable Naming Conventions

Stimuli, responses, and state all initially have

1) An identifier (SXXI, RXXI, TXXI,.... where XX defines the data object more specifically and I is a number),

2) A descriptive name (which is a textual description of the data object).

The stimulus, response or state will change to a specific variable name at some time before the design is implemented. When that change happens, which could be at various design steps of the box structures algorithm, the design object should have the following appear:

```
[YXXI: Descriptive name]
concrete_name : type;
```

In this way, the transition between the original data object name, and a subsequent/final name is clear and unambiguous. If a subsequent name is changed (made more concrete), the following will appear:

```
[concrete_name :    type;]
more_concrete_name: type;
```

One will need to make sure that uses of a new variable are consistent just as when verifying that state data is equivalent to stimulus history.

End of Variable Naming Conventions.

It is important to employ an effective means of recording the black box information. Template 1 provides a sample table for recording the stimuli and responses of a black box.

Template 1: Black Box Stimuli and Responses for Box <boxname>

List stimuli and responses for the box <boxname>.

stimuli

S1: Invocation
SF2: Sensor value(sensor reading)
<S3>: <s3>
<S4>: <s4>

responses

RF1: Open file for input
RF2: Read value
<R3>: <r3>
<R4>: <r4>

Design Notes:

RF1 and RF2 are commands sent to the File.

- In developing stimuli and responses it is necessary to adopt the view of the software that will be receiving the stimuli and issuing the responses.

Helpful Hints:

- It is often useful to have a naming convention for stimuli and responses. For example, having the letter(s) after the S or R define the external black box/device that the stimulus comes from or response goes to.

- When there is difficulty in coming up with proper black box behavior, it is often useful to go back to the stimuli and responses, to ensure that all are complete.

- Always keep in mind the invocation stimulus and the clock pulse stimulus, which are in many systems and are often ignored.

4.1.2 Task 2 - Develop Black Box Behavior

Stimuli and responses are combined into black box functions to describe the behaviors of the system. Template 2 provides a box description language (BDL) format for defining black box functions. The internal BDL (denoted by <>) is described by the BDL BNF found in Exhibit A of this manual. Line numbering conventions for the BDL are found in Exhibit B.

It is important to keep in mind that a black box can be reorganized a number of times before moving on to a later step. In a reorganization, the behavior and level of abstraction (black, state or clear box) are kept, but the form of the black box is modified to make it more readable for the development team. For reorganizations, identical behavior needs to be confirmed, so the same analysis and verification task is necessary, although it is often much more simple, since there is no transformation.

Template 2: Black Box Function for Box <boxname>

begin black box function S* || S: <boxname>

black box sub-function S* || S1: Invocation is
B01.01 RF1: Open file for input;
B01.02 RF2: Read value;
XOB;

black box sub-function <S2>: <s2> is
<>
XOB;

end black box function S* || S: <boxname>

Design Notes:

- Given the stimuli and responses identified in Template 1 it is straightforward to write down each of the responses in terms of stimuli histories.

- In general, the steps of discovering stimuli, responses, and behaviors are performed concurrently. One first develops some stimuli and responses and then begins to refine the Black Box function. This leads to identification to the need for more stimuli and/or responses. This cyclic process continues until it seems the definition is complete. Then one moves on to examine and to verify the function. These tasks may uncover the need for more stimuli and/or responses or modifications to the control structure for the subfunctions.

Helpful Hints:

- If the black box is hard to read, but expresses correct behavior, reorganize the black box

- Behavior must be in terms of stimuli histories, all conditions, etc. must be checked to ensure that all behavior is only in terms of stimuli histories.

4.1.3 Task 3 - Examine and Verify Black Box

Black box analysis and verification evaluates the quality and completeness of the black box specification. For the black box, the following types of analyses can be performed:

Black Box Closure - Closure analysis would ensure that all stimuli, responses, and behaviors are necessary and sufficient in the system. A straightforward algorithm for checking black boxes closure is:

Black Box Closure Algorithm

Given:

$S = (s_1, s_2, \ldots, s_n)$: complete set of stimuli entering the system

$R = (r_1, r_2, \ldots, r_m)$: complete set of responses generated by the system

$F = (f_1, f_2, \ldots, f_p)$: complete set of subfunctions describing the behavior of the black box

Step 1: Check that all responses are generated:

For all r_j in R there exists a subset S_A of S and a f_k in F such that $f_k(S_A) \rightarrow r_j$. In other words, ensure that each response results from at least one stimulus subfunction.

Step 2: Check that all stimuli are used:

For all s_i in S, there exists an S_A where $s_i \in S_A \subseteq S$, and there exists a r_j in R and f_k in F, such that $f_k(S_A) \rightarrow r_j$ and $f_k(S_A - s_i) \not\rightarrow r_j$. In other words, ensure that there is a stimulus subfunction for each stimulus.

Step 3: Check that all subfunctions are used:

For all f_k in F there exists a r_j in R such that $f_k(S_A) \rightarrow r_j$ where $S_A \subseteq S$. In other words, ensure that all stimulus response pairs exist.

End of Black Box Closure.

Black Box Completeness - Via reviews with customers, users, managers, and domain experts (for the first level black box) or amongst team members (for lower level black boxes), the Development Team must validate that all system requirements are captured in the Black Box Design.

Black Box Clarity - The Black Box Design must be clearly understood by the entire development team. An analysis of clarity would focus attention on domain jargon, naming conventions, intended functions, and other sources of miscommunication.

Template 3 can be used to record these analyses along with other analyses performed on the black box definition.

Template 3: Black Box Analysis for Box <boxname>

List all analyses performed for the black box definition.

Hypothesis: Black box closure exists.

- Analysis Process:

- Results:

- Black Box Modifications:

Hypothesis: The black box is complete.

- Analysis Process:
  (1) Make a mapping between each line of the black box and a section of the problem description.
  (2) Ensure that all parts of the problem description have been covered.

- Results:

- Black Box Modifications:

Hypothesis: The black box is clearly described.

- Analysis Process:

- Results:

- Black Box Modifications:

Additional Analyses as Required

Helpful Hints:

- Black boxes that are not readable should be reorganized.

4.2 State Box Development

The state box of a system or subsystem expands the black box by identifying data at this system level to be stored between stimuli so that only a current stimulus is required but not previous history. Let T be a set of possible data states at the top level, and let t be the initial state of the system or subsystem. As noted above, the state box contains an internal data abstraction that is defined by another black box, say g. In this case, the internal black box has a compound stimulus consisting of the external stimulus and the internal state and a compound response consisting of the external response and the new internal state. That is, g has the form

$$g: (S \times T)^* \rightarrow (R \times T).$$

Then, each pair <t,g> of an initial state and an internal black box function will uniquely define the behavior of the system. Note that the internal data abstraction will be capable of maintaining more deeply stored data, with the internal black box using its compound stimulus histories.

The state box is verified consistent with the original black box by removing the state representation and recovering the black box function from stimulus history to response. The following tasks are performed to produce a black box design. The tasks are done in the order that most effectively results in the work getting done completely and correctly.

4.2.1 Task 4 - Determine State Data

The state of the system is created by encapsulating required stimulus history in a state box. Data design methods, such as Entity-Relationship models, are used to create a state design. A number of excellent textbooks exist on the topics of data structures, file organizations, and database modeling. Exhibit C demonstrates an example of database modeling using Entity-Relationship and relational models.

The first step of developing a state box is to identify effective state data to represent stimuli histories. There are many valid state representations. The skill and experience of the designer will help identify the most effective data design. The following template can be used to match stimuli histories to state data.

Template 4: State Data Design for Box <boxname>

In this step, all stimuli histories used in the black box are listed. The purpose is to be able to thoroughly consider all potential state data items T.

S1: Invocation
SD1: None

SF2: Sensor value(sensor reading)
SD2: number of inputs processed - handles hour/minutes (ie, number of input that have been read or processed) for any stimulus

hourly mean - handles hourly mean as well as malfunction

hourly violations - handles number of violations for an hour

number of high readings - handles five high readings = violation situation

S3: <>
SD3: <>

S4: <>
SD4: <>

Helpful Hints:

- Scan the black box and drag any observed stimuli histories into this section.
- Group related stimuli histories to begin determining what the state data candidates are.

4.2.2 Task 5 - Retain/Migrate State Data

In this step, a software engineer must decide what state data items will appear in the current level object. Typically, a trade study would be conducted in order to assess alternatives, such as impact of a particular configuration on usage or development. This is an analytical step, resulting in the major state box decision.

Data items should be grouped in objects (i.e., boxes) in order to provide coherent object meaning and clean hierarchical decomposition. State data maintained at the current level of the box structure usage hierarchy will be visible to all boxes below in the usage hierarchy. The decision

on data grouping will strongly influence the clear box decomposition defined later. The following template can be used to group state data to be maintained at this level.

Template 5: State Data to be Maintained in Box <boxname>

State data requirements for box <boxname>.

T1: number of inputs processed
T2: hourly mean
T3: <>
T4: <>

State data to be maintained at this level.

T1: number of inputs processed
T2: hourly mean (hourly means are handled for each of the 24 hours)

Rationale for selection.

State data to be migrated to lower levels.

Tx: <xname>
Ty: <yname>

Helpful Hints:

- This is a major decision step, a trade study is useful, as is bottom up reasoning.

4.2.3 Task 6 - Develop State Box Behavior

In this step, an engineer will develop the state box, by using state data as a surrogate for the replaced stimuli histories. The line numbering strategy is used to identify the new, changed, and unchanged design statements. That will be used to focus the verification strategy.

The definition of state box behaviors is driven by the state grouping decisions made in the previous step, since differences between the black and state boxes are strictly a function of the state data selected to be kept at this level. The major decision for the state box is the state grouping, leaving the definition of the state box as the creative step that will document the results of the decision. The following template records the state box. The internal BDL (denoted by

<>) is described by the BDL BNF found in Exhibit A of this manual. Line numbering conventions for the BDL are found in Exhibit B.

It is important to keep in mind that a black box can be reorganized a number of times before moving on to a later step. In a reorganization, the behavior and level of abstraction (black, state or clear box) are kept, but the form of the black box is modified to make it more readable for the development team. For reorganizations, identical behavior needs to be confirmed, so the same analysis and verification task is necessary, although it is often much more simple, since there is no transformation.

Template 6: State Box Behavior for Box <boxname>.

Development tasks:

(1) copy black box function

(2) edit to develop required sub-functions

(3) upon completion of step, renumber subfunction statements

State Box BDL:

begin state function S: <boxname>

state box sub-function S1: Invocation is

```
        S01.01  con
        S01.02      number_processed := 0;
        S01.03      mean(0..23) := 0;
        S01.04      violations(0..23) := 0;
        S01.05      high_reading := 0;
B01.01  B01.06      RF1: Open file for input;
        S01.07  noc;
B01.02  S01.08  RF2: Read value;
        XOB;
```

state box sub-function <S2>: <s2> is

```
                <>
        XOB;
```

end state box function S: <boxname>

Helpful Hints:

- Replace all stimuli histories for which you have decided to keep state data at this level as a surrogate.
- Make sure that line numbering is correct, since it will facilitate the verification step.
- Reorganize hard to read designs. That will facilitate verifications.

4.2.4 Task 7 - Examine and Verify State Box

State box analysis and verification evaluates the quality and completeness of the state box design. Types of state box analyses/verification include:

State Box Closure - Closure analysis would ensure that all stimuli, responses, state, and behaviors are necessary and sufficient in the system. A straightforward algorithm for checking state box closure is:

State Box Closure Algorithm

Given:

$S = (s_1, s_2, \ldots, s_n)$: complete set of stimuli entering the system

$R = (r_1, r_2, \ldots, r_m)$: complete set of responses generated by the system

$T = (t_1, t_2, \ldots, t_r)$: complete set of state data items encapsulated in the system

$G = (g_1, g_2, \ldots, g_p)$: complete set of subfunctions describing the behavior of the black box

Step 1: Check that all responses are generated:

For all r_j in R there exists a subset S_A of S, a subset T_A of T, and a g_k in G such that $g_k(S_A, T_A) \rightarrow r_j$

Step 2: Check that all stimuli are used:

For all s_i in S, there exists an S_A where $s_i \in S_A \subseteq S$, and there exists a r_j in R, a T_A in T, and g_k in G, such that $g_k(S_A, T_A) \rightarrow r_j$ and $g_k(S_A - s_i, T_A) \not\rightarrow r_j$

Step 3: Check that all state data are used:

For all t_i in T, where $t_i \in T_A \subseteq T$, there exists a r_j in R, a $S_A \subseteq S$, and a g_k in G such that $g_k(S_A, T_A) \rightarrow r_j$ and $g_k(S_A, T_A - t_i) \not\rightarrow r_j$

Step 4: Check that all subfunctions are used:

For all g_k in G there exists a r_j in R such that $g_k(S_A, T_A) \rightarrow r_j$ where $S_A \subseteq S$ and $T_A \subseteq T$

End of State Box Closure.

State Box Completeness - Via team reviews, the Development Team must validate that all system requirements are captured in the State Box Design.

State Box Clarity - The State Box Design must be clearly understood by developers. An analysis of clarity would focus attention on domain jargon, data naming conventions, and other sources of miscommunication.

State Migration Analysis - The Development Team performs an analysis on the quality of their state migration decisions performed in Task 6.

State Usage Analysis - An important analysis procedure is to record the state data usage in the state box. All references to each data item is identified and recorded. The record of data usage can be used in a number of ways for the analysis of data flow in structure programs. A procedure for recording state data usage is:

(1) copy names of state data being maintained at this level
(2) for each state data record all required references in the state box subfunctions

State Maintenance Analysis - It is critical to ensure that all state data maintenance is done correctly. In that manner, one can prove that the state data serving as a surrogate for stimulus history in the black box reflects the same behavior. The procedure for recording state maintenance is:

(1) Ensure all statements that have been changed from the Black Box to the State Box exhibit the same behavior. Show that each pair of different statements generate the same (stimuli, response) pairs. Engineers proceed by listing each pair of different statements and then following the pair with an argument that both statements generate the same set of (stimuli, response) pairs for the entire domain of operations.

(2) Ensure that all new state box statements are used in the above arguments. This is done to ensure that all new statements introduced in the State Box are required to show equivalence, meaning that behavior was not modified. If extra statements are left in the function they will cause the state data to take on the wrong value at some time in calculating the values for the state data. This check is performed by listing each State Box statement in a column. In a second column we insert the fact that the statement is a key word or indicate what proof argument(s) for which the State Box statement was used to show equivalence. Any statements listed in column 1 that do not have a corresponding entry in column 2 are extra statements that are not required.

(3) Confirm that the control structure remains unchanged for each black box statement (new or modified). This is done by confirming that the conditions which would lead to each black box statement have remained unchanged. To do this, an engineer must confirm that all control flow statements a black box statement is nested within do not modify the behavior of the black box statement in the following four cases:

1) Structures that the black box statement is nested within must be in a state box equivalence argument.
2) Black box statement must be in same position in a sequence structure relative to other Black box statements.
3) Black box statement must not be in a new state box iteration or alternation structure.
4) Black box statement must not be in a new state box concurrent structure with more than one Black box statement.

The following template can be used to record the analyses performed on the state box design.

Template 7: State Box Analysis for Box <boxname>

List all analyses performed for the state box design.

Hypothesis: State box closure exists.

- Analysis Process:

- Results:

- State Box Modifications:

Hypothesis: The state box is complete.

- Analysis Process:

- Results:

- State Box Modifications:

Hypothesis: The state box is clearly described.

- Analysis Process:

- Results:

- State Box Modifications:

Hypothesis: State migration is correct.

- Analysis Process:

- Results:

- State Box Modifications:

Hypothesis: State usage is complete.

- Analysis Process:

  State Data Usage for box <boxname>

  ```
  HIGH_READING
  S01.05  high_reading := 0;
  S02.07  high_reading := 0;
  S02.14  high_reading := high_reading + 1;
  S02.19  high_reading := 0;
  S02.23  if high_reading = 5
  S02.31  high_reading := 0;
  ```

  End of State Data Usage Algorithm.

  State Data: Lower Levels

  [(1) copy names of state data to be maintained at lower levels from step 5 and (2) then for each state data record all required references]

- Results:

- State Box Modifications:

Hypothesis: State maintenance is correct.

- Analysis Process:

1)

State box subfunction SF2: Sensor value(sensor reading)

Statement pairs and equivalence arguments:

Hypothesis - S02.37 exhibits the same behavior as B02.01

statement pair: B02.01/S02.37

S02.37 **if** number_processed = 1440

B02.01 **if** 1440 SF2 stimuli have been received since most recent S1 stimulus

equivalence argument:

number_processed keeps the running count of stimuli read which, in effect, handles the hours and minutes of the readings, since each sensor value represents the reading at another minute. number_processed is initialized at invocation (S01.02). It is incremented after each sensor value is processed (S02.36), but before the number_processed is checked for completion (ie, 1440 readings read).

2)

New S statements:	**Used in equivalence argument / keyword:**
S01.01	Keyword
S01.02	B02.01/S02.36, B02.04/S02.39, B03.02/S03.02
S01.03	B02.04/S02.39, B03.02/S03.02

3)

Black Box Stmt:	In equiv. arg.:	Seq:	I/A:	Con:
B01.06	N/A	T	N/A	N/A
S01.08	N/A	T	N/A	N/A
S02.37	N/A	T	N/A	N/A
B02.38	T	N/A	T	N/A

- Results:

- State Box Modifications:

Additional Analyses as Required

Helpful Hints:

- Only verify what is new or has been changed. Verifications are a function of the extent of change in the design, not a function of the size of the design.

- Analysis and verification by direct assertion is preferable. If a design is hard to read, redesign.

4.3 Clear Box Development

A state box can be expanded into a clear box by replacing the internal data abstraction with a procedural structure of new data abstractions in either sequential or concurrent logic. Sequential structures may involve simple sequence, alternation, or iteration whose semantics are well known from sequential programming. Since sequential programs are rules for mathematical functions, from initial states to final states of computation, a clear box in sequential structures defines the functional behavior in terms of the next level black boxes. Concurrent structures require more analysis and discipline in use because of their potential complexities.

The clear box development produces a top-down, stepwise refinement of the increment functionality. A usage hierarchy of box structures is constructed during system design via the application of both system decomposition and object composition. Top-down system decomposition enables an essential intellectual control in development. The system grows one level at a time. The mathematical structuring of systems in usage hierarchies of objects allows formal verification methods to be used. Also, the referential transparency of objects in a clear box provides an essential modularity and design independence to each object. The internal black boxes are defined on the next level of a box structure usage hierarchy, as shown in Figure 1.

Clear box designs are verified to the original state by eliminated the procedural structure and merging all internal black boxes into a single internal data abstraction. Sequence and alternation structures are eliminated by function composition and disjoint union directly. Iteration structures can be reformulated as recursion, but iteration free. In this way, clear box designs can be verified against state box specifications.

4.3.1 Task 8 - Identify Data Objects and Internal Black Boxes

The identification of objects (i.e., boxes) at the next level of the system hierarchy is a key task. The state grouping analysis performed in the state box definition provides some guidance here. State data not maintained at this system level will be migrated downward to lower level objects. The grouping of this lower level state into data abstractions will define black boxes within the current clear box.

There are two sub-tasks in Task 8. Task 8A defines the data abstractions for the state data to be maintained in the clear box at this level of the hierarchy. The state data are completely designed and described in terms of data type and length. Task 8B then deals with the data

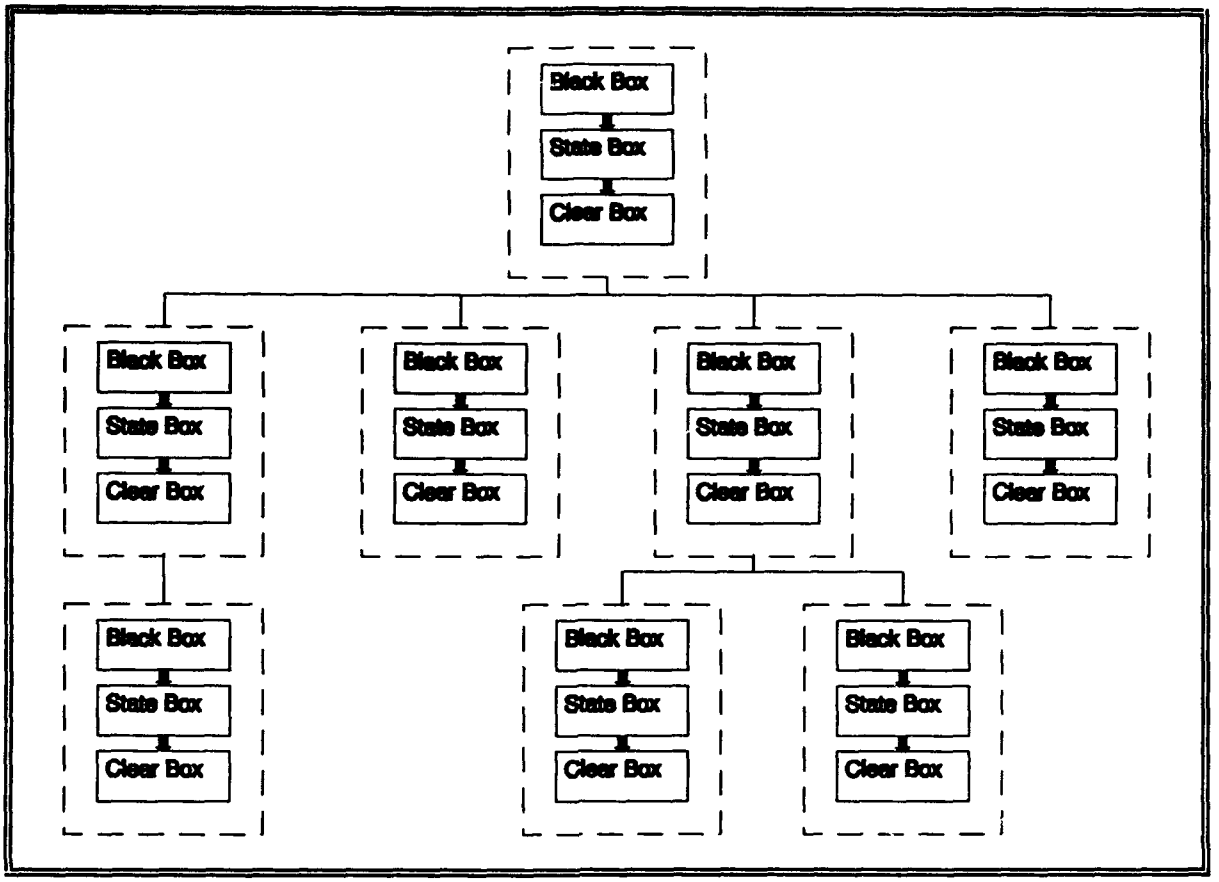


Figure 1: Box Structure Usage Hierarchy

The following template can be used to record both steps 8A and 8B. The internal black boxes are defined fully in 8B.

Template 8: Internal Black Boxes for Box <boxname>

Task 8A: Define data abstractions for state data at this level.

T1: password = String of X characters

T3: buoy_status = [shutdown, restart, active]

Design Notes:

- The limit of password size will be finalized upon discussion with other buoy designers.

Task 8B: Define communications with lower level black boxes for migrated state data.

In creating the clear box, a number of stimuli to lower lever black box need to be invoked to handle stimuli histories.

In the Buoy case study, the following stimuli are defined:

SOS_status
broadcast_status
broadcast_waiting

In all cases it will be necessary to set and get each of the 11 state values. In the case of the four data values it will be necessary to put, get and clear values. Stimuli to accomplish putting, getting and clearing these values are:

set_SOS_status(X) - which sets the SOS status to X (X=ON/OFF)

get_SOS_status - which gets the present SOS status

set_broadcast_status(X) - which sets the broadcast status to X (X=ON/OFF)

get_broadcast_status - which gets the present broadcast status

set_broadcast_waiting(X) - which sets the broadcast waiting status to X (X=TRUE/FALSE)

get_broadcast_waiting - which gets the broadcast waiting status

---

Helpful Hints:

- Consider all possible use of migrated state data. That will facilitate the determination of proper stimuli to that state data.
- Use abstract data types (Chapter 3 of Structured Programming) to define complex data items at this level.

4.3.2 Task 9 - Develop Clear Box Behavior

Given the internal black box behaviors, the clear box defines the procedurality and communication among the black boxes. The following template provides a box description language (BDL) for recording the clear box design. The internal BDL (denoted by <>) is described by the BDL BNF found in Exhibit A of this manual. Line numbering conventions for the BDL are found in Exhibit B.

Template 9: Clear Box Function for Box <boxname>

```
begin clear box  <boxname>

stimuli:
   <copy from Template 1>
   S1:  Invocation
   SF2: Sensor value(sensor reading)
responses:
   <copy from Template 1>
   RF1: Open file for input
   RF2: Read value
state:
   <copy from Template 5>
   T1:  number_processed = INTEGER
   T2:  mean = Array (0..23) of REAL
data variables:
   i :       INTEGER
   <>
proc <boxname>
(1) copy state box function
(2) edit to develop required procedure
(3) upon completion of step, renumber clear box statements
S01.01  S01.01  con
S01.02  S01.02      number_processed := 0;
                    [initialize mean and violations arrays]
        1.03        for
                       i := 0 to 23
        1.04        do   []
S01.03  1.05           mean(i) := 0;
S01.04  1.06           violations(i) := 0;
        1.07        od;
S01.05  S01.08      high_reading := 0;
B01.06              [RF1:   Open file for input;  --->]
        1.09        open(file);
S01.07  S01.10  noc;
.
.
.

corp <boxname>
```

Design Notes:

- In this step, an engineer develops the clear box, defining the behavior in terms of a process. The line numbering strategy is used to identify the new, changed, and unchanged design statements. That will be used to focus the verification strategy.

- The final, desired clear box must be considered before the first clear box is created. This means issues concerning the final source code version should be thought about now. In this manner, clear boxes will be created in a logical manner that will ease future refinements and verifications towards the final clear box. Not thinking about the final clear box may lead to illogical designs and unverifiable code.

Helpful Hints:

- The control structure for the clear box is the major decision at this level.

- Keeping to the line numbering approach will facilitate verifications.

- If parts of the clear box are unreadable, reorganize. That will also facilitate verification.

4.3.3 Task 10 - Examine and Verify Clear Box

Clear box analysis and verification evaluates the quality and completeness of the clear box design. Types of clear box analyses/verification include:

Clear Box Closure - Closure analysis would ensure that all stimuli, responses, state, internal black boxes, and procedural structures are necessary and sufficient in the system. A similar algorithm to state box closure is used with additional consideration of the internal black boxes and the procedural structures.

Clear Box Completeness - Via team reviews, the Development Team must validate that all system requirements are captured in the Clear Box Design.

Clear Box Clarity - The Clear Box Design must be clearly understood by customers as well as developers. An analysis of clarity would focus attention on domain jargon, data naming conventions, procedural structure (e.g., pseudo-code, flowcharting) conventions, and other sources of miscommunication. This is only a concern for the top level clear box.

State Migration Analysis - Based upon the design of the internal black boxes, another state migration analysis is performed to evaluate the state migration decisions.

Referential Transparency - The procedural clear box design ensures that each internal black box is referentially transparent from all other peer black boxes and common services in the clear box. Thus, each black box can be designed independently.

Procedure Correctness Analysis - The procedural clear box design must be correct with regards to the procedure free behavior described in the state box. The steps to confirming the clear boxes' correctness are the following:

(1) Ensure all statements that have been changed from the State Box to the Clear Box exhibit the same behavior. We need to show that each pair of different statements generate the same (stimuli, response) pairs. Engineers proceed by listing each pair of different statements and then following the pair with an argument that both statements generate the same set of (stimuli, response) pairs for the entire domain of operations.

(2) Ensure that all new Clear box statements are used in the above arguments. This is done to ensure that all new statements introduced in the Clear Box are required to show equivalence, meaning that behavior was not modified. If extra statements are left in the function, then the Clear box implements a function with behavior different than that found in the State Box. This check is performed by listing each Clear Box statement in a column. In a second column we insert the fact that the statement is a keyword or indicate what proof argument(s) for which the State Box statement was used to show equivalence. Any statements listed in column 1 that do not have a corresponding entry in column 2 are extra statements that are not required.

(3) Confirm that the control structure does not affect behavior for each State box statement (new or modified). This is done by confirming that the conditions which would lead to each State box statement have remained unchanged. To do this, an engineer must confirm that all control flow statements a State box statement is nested within do not modify the behavior of the State box statement.
   1) Structures that the state box statement is nested within must be in a Clear box equivalence argument.
   2) State box statement must be in same position in a sequence structure relative to other State box statements.
   3) State box statement must not be modified by a new Clear box iteration/alternation structure, or a removed structure.
   4) State box statement must not be in a new Clear box concurrent structure with more than one State box statement and removed concurrent structures must also have no effect.

Reuse Analysis - The design of internal black boxes should include a thorough consideration of reuse opportunities. The ability to reuse design objects within and among systems provides tremendous productivity and quality advantages. Pre-existing design objects are stored on an organizational repository. An object requirement, stated as a black box, can be matched with existing object classes stored for reuse. During an analysis of the black box the benefits and costs of object reuse and modification can be studied. Knowledge of existing object classes or

insight into desired object classes will influence the designer's invention of data abstractions as black boxes at the next level in the box structure hierarchy.

As an example of matching black box behavior with reusable objects, assume there exists a reusable software module with black box transaction behavior, $r(I'^*) \rightarrow O'$, where I' and O' are the inputs to and outputs from the module. Given a black box transaction, $p_i(I''^*) \rightarrow O''$, we are able to evaluate the potential for the reusable module to match the black box behavior. Requirements matching must be done on inputs (I' and I"), outputs (O' and O"), and behaviors (r and p_i). If an exact match is not found, several alternatives can be studied:

1. Use the reusable module as is and modify the black box behavior and the clear box to accommodate its behavior.

2. Modify the behavior of the reusable module to match the black box behavior.

3. Modify both the behavior of the reusable module and the black box in order to produce an effective match.

4. Do not use the reusable module and search for other reuse opportunities or decide to develop a module from scratch to satisfy the system requirement.

A detailed matching algorithm is needed, along with a cost tradeoff procedure to evaluate the most effective reuse strategy.

Common Service Analysis - The design of internal black boxes should include a thorough analysis of common services in the development environment. As discussed in Section 3, common services are well-defined procedures that are reused within and among systems. Common services are pre-defined functions and procedures that are run from the clear box with appropriate input and output parameters. Examples of common services are mathematical routines, interfaces with external devices (e.g., printers), database systems, etc.

Concurrency Analysis - The clear box design should be analyzed for effective use of concurrency. Opportunities for concurrency should be identified and maximized in the design. Implementation decisions will determine whether the full concurrency of the system can be realized in the implementation environment.

The following template can be used to record the analyses performed on the state box design.

Template 10: Clear Box Analysis for Box <boxname>

List all analyses performed for the clear box design.

Hypothesis: Clear box closure exists.

- Analysis Process:
- Results:
- Clear Box Modifications:

Hypothesis: The clear box is complete.

- Analysis Process:
- Results:
- Clear Box Modifications:

Hypothesis: The clear box is clearly described.

- Analysis Process:
- Results:
- Clear Box Modifications:

Hypothesis: State migration is correct.

- Analysis Process:
- Results:
- Clear Box Modifications·

Hypothesis: The clear box is correct with regards to the state box.

- Analysis Process:

1)
Hypothesis - 1.05 exhibits the same behavior as S01.03

statement pair: S01.03/1.05
1.05 mean(i) := 0; AND
S01.03 mean(0..23) := 0;

equivalence argument:
Lines 1.03, 1.04, 1.07 serve as a loop to substitute a instantaneous array initialization with a sequential one, which is done in reality.

hypothesis confirmed

2)

New C statements:	**Used in equivalence argument / keyword:**
1.03	S01.03/1.05, S01.04/1.06
1.04	S01.03/1.05, S01.04/1.06

3)

State Box Stmt:	In equiv. arg.:	Seq:	I/A:	Con:
1.05	T	N/A	T	N/A
1.06	T	N/A	T	N/A
S02.14-S02.15	N/A	N/A	T	N/A

- Results:

- Clear Box Modifications:

Hypothesis: Referential transparency exists.

- Analysis Process:

- Results:

- Clear Box Modifications:

Hypothesis: Reuse analysis is complete.

- Analysis Process:

- Results:

- Clear Box Modifications:

Hypothesis: Common service analysis is complete.

- Analysis Process:
- Results:
- Clear Box Modifications:

Hypothesis: Concurrency analysis is complete.

- Analysis Process:
- Results:
- Clear Box Modifications:

Additional Analyses as Required

Helpful Hints:

- Verify only what's been changed. Verifications are a function of the extent of change, not of the size of the clear box.
- If portions of the clear box are hard to examine or verify, reorganize the clear box.

4.4 Clear Box Refinements

The initial clear box design will nearly always undergo a series of refinements based upon subsequent design decisions made in the development of the current increment. Analyses that would lead to refinements include the optimization of control structures and further consideration of common service subsystems and reusable objects as alternatives for the behaviors of the internal black boxes in the clear box.

4.4.1 Task 11 - Refine Clear Box

The initial clear box design may be refined one or more times based upon required system changes, new and improved understandings of system solutions, and effective common service and reusable object opportunities. Each new refinement must be carefully recorded and verified as correct. Template 11 provides a format for recording each refinement.

Template 11: Clear Box Refinement for box <boxname>

Refinement <refnum>: Clear Box Function Refinement for box <boxname>.

begin clear box <boxname>

proc <boxname>

(1) copy the previous clear box function

(2) edit to develop required procedure

(3) upon completion of step, renumber clear box statements

.
.
.

```
S01.08  S01.08     high_reading := 0;
        B01.06     [RF1:   Open file for input;  --->]
1.09    1.09       open(file);
S01.10  S01.10     noc;
S01.11             [RF2:   Read value;  --->]
1.11    1.11       read(stimulus);
        2.12       hour := number_processed DIV 60;
        2.13       minute := number_processed MOD 60;
```

.
.
.

corp <boxname>

Design Notes:

- In this step, an engineer refines the clear box, defining the behavior in terms of a process. The line numbering strategy is used to identify the new, changed and unchanged design statements. That will be used to focus the verification strategy.

- The final, desired clear box must be considered before the clear box is refined. This means issues concerning the final source code version should be thought about now. In this manner, clear boxes will be created in a logical manner that will ease future refinements and verifications towards the final clear box. Not thinking about the final clear box may lead to illogical designs and unverifiable code.

Helpful Hints:

- Small design refinements are the key. Making large logical leaps will lead to difficult verifications.

- If portions of the clear box are hard to examine or verify, reorganize the clear box.

4.4.2 Task 12 - Examine and Verify Refined Clear Box

Appropriate analysis and verification activities are performed based upon the types of refinements performed on the clear box. For example, if a control structure is altered, a clarity analysis should analyze the effect on the understandability of the new design structure. Additionally, it must be shown that the control structure is correct with regards to the previous clear box. If a new reusable object is identified and integrated into the clear box, then a new state migration analysis is needed, along with correctness confirmation. Thus, the template for analyzing the refined clear box would be identical to Template 12 for clear box analysis. However, only essential analyses or verifications would be performed.

Template 12: Refined Clear Box Analysis for Box <boxname>

List all analyses needed for the refined clear box design.

Hypothesis: Clear box closure exists.

- Analysis Process:

- Results:

- Clear Box Modifications:

Hypothesis: The clear box is complete.

- Analysis Process:

- Results:

- Clear Box Modifications:

Hypothesis: The clear box is clearly described.

- Analysis Process:
- Results:
- Clear Box Modifications:

Hypothesis: State migration is correct.

- Analysis Process:
- Results:
- Clear Box Modifications:

Hypothesis: The clear box refinement is correct with regards to the parent clear box.

- Analysis Process:

Hypothesis - 3.25 exhibits the same behavior as S02.20-S02.30 and S02.32-S02.35

statement pair: S02.27-S02.30,S02.32-S02.35/3.25

```
S02.27  con
S02.26      [Combine sensor reading with mean(number_processed DIV 60);  --->]
2.28        mean(hour) := mean(hour) + stimulus.sensor reading/60;
S02.30  noc;
S02.32  con
S02.31      [Combine sensor reading with mean(number_processed DIV 60);  --->]
2.33        mean(hour) := mean(hour) + stimulus.sensor reading/60;
S02.35  noc; AND

3.25        mean(hour) := mean(hour) + stimulus.sensor reading/60;
```

equivalence argument:
Since the exact functionality is done to mean(hour) in both cases of the if-then-else condition, it was decided that it would be better to have the code only appear once before the if-then-else. As a result, there is no need for the two con-noc structures, since an order has now been placed on the code.

hypothesis confirmed

- Results:
  - Clear Box Modifications:

Hypothesis: Referential transparency exists.

- Analysis Process:
- Results:
- Clear Box Modifications:

Hypothesis: Reuse analysis is complete.

- Analysis Process:
- Results:
- Clear Box Modifications:

Hypothesis: Common service analysis is complete.

- Analysis Process:
- Results:
- Clear Box Modifications:

Hypothesis: Concurrency analysis is complete.

- Analysis Process:
- Results:
- Clear Box Modifications:

Additional Analyses as Required

Helpful Hints:

- Strive for designs that are verifiable by direct assertion.

4.4.3 Repeat Step IV until Clear Box Completed

Clear box refinements are continued until the Development Team is satisfied with the quality and the correctness of the designed clear box. Each clear box refinement is kept small in order to make the required analyses and verifications as simple as possible.

4.5 Design Translation

Given a final clear box refinement, the software engineer produces the translation of the design in a appropriate form. The software component of the implementation is coded in an appropriate programming language. The resulting code is verified as consistent with the clear box design.

4.5.1 Task 13 - Design Translation

System implementation accepts the design specification in the form of a box structure usage hierarchy and provides the capabilities and resources to implement it. Implementation may be an integration of hardware, software, and human behavior. Implementation objectives are to build and optimize the specified system and to prepare users and operators for its operation and maintenance.

Software translation is performed under the rigors of structured programming as exemplified. The appropriate programming language is selected based on the system environment and requirements.

The Box Structures approach is programming language independent, but it can be language dependent if that is more efficient. The use of BDL is recommend, although it is certain that the target language for the software system will not be BDL. If one uses BDL, there is a transformation during the software implementation, where the developer moves from BDL to the target language. It is recommend that this transformation be done at the final refinement. Of course, the shift to the target language can be done at any time. Conceivably, that transformation may be done at the first black box. The critical notion is that language structures used must not be a superset of BDL. Final structures in many languages may differ from the syntax for BDL. Another well-defined syntax may be used, given the condition that only the same structure types are used.

BDL has been found to be useful because it is simple. It is also easy to provide automatic support for the transformation. An example for c is discussed below. The transformation must occur at some point in the project. When the transformation occurs, it should be viewed as a reorganization and not a refinement. One should just do the switch from BDL to target language without refining the design. That simplifies reading of the transformation. If there needs to be refinement, the previous refinement must appear as functional commentary. That will make the refinement easier to understand.

BDL is not a completely fixed language. The structures defined in the BDL are fixed, but there can be extensions to the language in a particular application domain. For example, if stimuli, responses or assignment have a particular syntax, using that syntax may simplify work. BDL is effective because it is simple, which makes it tailorable and extendable. It only requires that one limits the structures used, without limiting functionality.

A convention currently followed for using BDL and the target language is presented below.

One should use BDL for as many levels of design as possible. The syntax is simple and readable, which means that individuals reading a design will not need to acquire a programming language expertise in order to be productive. Where possible, only the final refinement should be converted to the target language, after all, only the final code is to be compiled. The final transformation to executable code should be done, if at all possible, using global searches and replaces with the editor at hand.

For example, a conversion of BDL into C might occur as in Template 13 below.

Template 13: Software Translation

BDL	C
if x = y	if x == y
then	{
z := x;	z = x;
else	}
z := y * 3;	else
fi;	{
	z = y * 3;
	}

If one looks carefully, they can determine the set of replacements that an editor, such as Word Perfect, would need to complete in order to correctly shift from BDL to code. Having either a script, a translator or a global definition file is the approach to create a correct translation from BDL to programming language code. More specifically, the following sequences of replacements would be done:

Item:		Replaced by:
=	----->	==
:=	----->	=
then	----->	{
else	----->	}
		else
		{
fi;	----->	}

Helpful Hints:

- This step should only be a translation, with no refinement.
- The primary intellectual challenge for this level of refinement is insuring that the translated design has proper data typing for the target language.

4.5.2 Task 14 - Verify Implementation

Each form of system component is verified correct and consistent with the final clear box design. The software code is verified by proving the equivalence of the structured program and the box-structured design found in the final clear box refinement. Template 14 provides a format for recording these arguments.

Template 19: Software Code Verification

Item:		Replaced by:	Correctness argument:
=	----->	==	
:=	----->	=	
then	----->	{	
else	----->	}	
		else	
		{	
fi;	----->	}	

Upon completion of the final software verifications, the box structure is sent to a Team Review for sign-off of all Completion Conditions as described in Section 2.2 of this manual.

Helpful Hints:

- This verification should be trivial, with the only intellectual challenge being the confirmation of the data typing.

4.5.3 Repeat Box Structure Algorithm for Each Internal Black Box

The box structure design algorithm is recursive in that each internal black box in the final clear box is designed by the same steps. A box structure usage hierarchy is constructed for the current increment j. The algorithm is complete when no further black box subsystems are identified in the design that are part of the current design increment.

When the increment is complete, then the code is given to the certification team for compilation, linking and execution. The development team can then begin to work on the next increment. At that time, the only responsibility that the development team has for the just completed increment is to isolate and resolve any failures observed by the certification team.

Section 5: Correct Test Increment (1...j)

The Cleanroom Certification Team is responsible for process P6.j, Software Certification for Increment j. This process is executed upon completion of process P5.j, Software Development and Certification Development for Increment j. The Cleanroom Development Team is responsible for correcting failures as they are found in the certification process. Specifically, task D6.j.4: Correct failure, verify correction and prepare ECN; defines the process of correcting failures in the increment design and returning the corrected increment j to the certification team. Below is found a lower level algorithm for correcting failures.

```
proc D6.j.4: Correct failure, verify correction, and prepare ECN of Increment j
   [This process results in a refined clear box and Engineering Change Notices to describe the
   corrective actions for increment j]
   do
      for all failures found by Certification Team
      do
         do
            D6.j.4.1 Isolate failure;
            D6.j.4.2 Correct failure;
            D6.j.4.3 Verify failure correction;
         until
            Correction is verified
         od;
         D6.j.4.4 Prepare Engineering Change Notice (ECN);
      od;
      D6.j.4.5 Hold team review for corrections;
      if Completion Conditions satisfied
      then
         D6.j.4.6 Submit test increment (1...j) to certification team;
      fi;
   od;
corp;
```

The certification team will complete Failure Reports on the current test increment (1...j). At designated points, as discussed in the *Certification Process Manual*, the existing Failure Reports will be accumulated and passed on to the Development Team for correction. The certification process will be suspended until the corrections are completed. The Certification Team does not make corrections to the test increment; that is the responsibility of the Development Team.

The above process model shows that Failure Reports are handled by isolating the failure, correcting it, and verifying the corrections. Then an ECN is prepared. Once all failures are corrected a team review is held. If all Completion Conditions are satisfied, the corrected test

increment (1...j) is returned to the Certification Team. The individual development tasks for making corrections are described here.

5.1 Isolate Failure

The Certification Team provides a formal Failure Report for each software failure found during testing. The Failure Report includes the testing scenario, the test data, and the test results. The format of the Failure Report is described in Volume 6. The Development Team should isolate the erroneous code. If the failure cannot be isolated, the certification engineers will be called in to work with the developers to determine if a failure exists.

Once the failure is found and isolated the development team begins corrective actions on the design and the code.

5.2 Correct Failure

Failure resolution may be a simple matter of correctly initializing a value or it may be a deep logic error that was not discovered during the development process. For simple errors, the needed modifications are made in the code and the increment design. In Cleanroom practice, typically a large majority of failures are caused by such simple errors.

For more complex failures, the software engineers must review all products of the development process (black box, state box, (refined) clear box, and code). Once the appropriate corrections are determined, the design trail should be modified to reflect the corrections. However, the actual design records and code should not be finalized until the corrections are verified.

In some cases, failure correction may require modifications to the designs of box structures at higher levels in the box structure hierarchy. A complete design trail for such complex failures must be recorded for thorough verification and team review.

5.3 Verify Correction

All corrections needed to resolve a Failure Report are verified as correct and consistent with the required software behavior. All levels of verification (black box, state box, refined clear box, and code) should be performed for each failure.

5.4 Prepare Engineering Change Notice (ECN)

A formal document, the Engineering Change Notice (ECN) is prepared to document all changes to the current increment. The modifications made to all design records and code in the box structure usage hierarchy are presented. The template for the ECN is shown below:

Template - Engineering Change Notice (ECN# _____)

Failure Report # _____

Date Received _____ Date Completed _____

Brief Description of Failure:_____

_____

_____

_____

Changes Made to Correct Failure:_____

_____

_____

_____

Design Objects Modified:

Black Boxes -

State Boxes -

Refined Clear Boxes -

Implemented Boxes -

Design Notes:

- One ECN may provide the corrections to satisfy one or more Failure Reports. If so, this should be noted and justified.

5.5 Team Review for Corrections

Once all Failure Reports are handled by a set of ECNs, the Development Team performs a team review to validate the correctness of all changes to the test increment (1...j). The team evaluates a set of Completion Conditions, such as:

Failure Corrections Completion Conditions

1. Have all Failure Reports been resolved?

2. Have ECNs been completed for all modifications to the design and code?

3. Is each ECN complete with full design trails for the corrections?

4. Is the current state of the box structure design records complete and up-to-date?

5. Have all pertinent reviews for this process been completed?

5.6 Submit Test Increment (1...j) to Certification Team

Upon successful completion of the team review the corrected test increment (1...j) is returned to the Certification Team for further testing. The Development Team turns over complete control of the design and code. No additional changes can be made until the next set of Failure Reports are received.

Exhibit A - BOX DESCRIPTION LANGUAGE BNF

The BNF below is presented to clarify the valid syntax for Black, State and Clear Boxes. More specifically, one will see that a part of the BNF that appears here for the Black, State and Clear boxes also appears in the respective templates for each box.

Notation: $^{+}$ means 1 or more
* means 0 or more

<design object> ::= <black box> | <state box> | <clear box>

<black box> ::= **black box function S* || S <object name> is**
[black box subfunction S*|| <stimulus name> is
<structure>$^{+}$
XOB;]$^{+}$
end black box function S* || S <object name>

<state box> ::= **state box function S: <object name> is**
[state box subfunction <stimulus name> is
<structure>$^{+}$
XOB;]$^{+}$
end state box function S: <object name>

<clear box> ::= **begin clear box <object name> is**
[stimuli
<stimulus name>*
responses
<response name>*
state
<state name>*
data variables
<variable name>* |
var
<stimulus name>*
<response name>*
<state name>*
<variable name>*]
[proc <object name>
<CBstructure>$^{+}$]*
corp;
|
begin clear box <object name> is
stimuli
<stimulus name>*

responses
<response name>*
package <object name>
state
<state name>*
data variables
<variable name>*
[**proc** <object name>
<CBstructure>+
corp;]+
egakcap;
end clear box <object name>

<object name> ::= *object name*

<stimulus name> ::= *SXXi: stimulus name* | <variable name>

<response name> ::= *RXXi: response name* | <variable name>

<state name> ::= *TXXi: state name* | <variable name>

<variable name> ::= *variable name*

<structure> ::= <sequence> | <fordo> | <ifthen> | <ifthenelse> | <case> | <whiledo> | <dountil> | <connoc>

<CBstructure> ::= <structure> | <run> | <use>

<sequence> ::= (<structure> | <NOOP> | <statement>;+)+

<fordo> ::= []
for
indexlist
do []
<sequence>
od;

```
<ifthenelse>  ::=  []
                   if
                      <condition>
                   then []
                      <sequence>
                   else []
                      <sequence>
                   fi;

<case>        ::=  []
                   case
                      p
                   ([]
                   part(CLi)
                      <sequence>)+
                   []
                   else
                      <sequence>
                   esac;

<whiledo>     ::=  []
                   while
                      <condition>
                   do   []
                      <sequence>
                   od;

<dountil>     ::=     []
                      do
                         <sequence>
                      until
                         <condition>
                      od;

<connoc>      ::=     []
                      con
                         <sequence>
                      noc;

<run>         ::=  run <service>

<use>         ::=  use <BBstatement>
```

<statement> ::= any statement that is returning a response, presenting a description of actions (non-commentary), or making an assignment.

<condition> ::= any logical expression that can evaluate only to true or false.

<service> ::= A name of a common service that is at the current level of the software system's parts hierarchy.

<BBstatement> ::= A stimulus to a lower level black box

<NOOP> ::= Either some sort of no operation statement, or a blank line.

Exhibit B - LINE NUMBERING RULES FOR BOX DESCRIPTION LANGUAGE

Since functional verification entails determining whether two items both define the identical function, each line of design can be viewed as a separate sub-function. If that line is not changed, its subfunction does not change. In that manner, once a line is verified to define the same function as a line in a previous design, it does not need to be verified again, unless it is changed.

As a result, the amount of verification is only a function of the number of changes and additions, not a function of the present size of the design. To handle verifications correctly, a line numbering strategy is needed.

Every non-comment line in a pseudo-code representation of a box will have one or two identifiers associated with it.

- The position of the line in the previous box
- The position of the line in the present box

Of course, lines in a black box do not have previous box identifiers associated with them, nor do new lines created in the present box.

The present box identifier has the format XXX.YYY where:

XXX identifies the box where the line was most recently modified (newly created lines have the present box identifier).

YYY identifies the present sequential position of the code in the box. The third digit is typically left blank, resulting in YYY having the valid range of 00-99.

The previous box identifier has the format XXX.YYY where (newly created lines, of course, will have not previous box identifier):

XXX identifies the box where the line was most recently modified previous to this box.

YYY identifies the sequential position of the code in the previous box. The third digit is typically left blank, resulting in YYY having the valid range of 00-99.

For a black box, the identifier XXX is B??, where ?? is a number representing the current black box subfunction.

For a state box, the identifier XXX is S??, where ?? is a number representing the current state box subfunction.

For a clear box, the identifier XXX is C?? where ?? is a number representing the clear box instance. For example, the first clear box is 1, its refinement is 2, and so on.

For clear box refinements, the syntax of the clear box is kept, with the exception as described below. Lines have intended functions that are refined and are now comments keep previous line number and have no present number, while the lines that are refined have just a present box number and no previous number.

As stated above, the last character of the YYY for the line number should be left blank. In that manner, if a change to a box is necessary after a verification, the amount of renumbering will be minimized by using the blank character for numbering inserted lines.

When the last character is being used, the values a-z should be used. In that manner, up to 26 lines can be inserted between two other lines as a correction. If more than 26 lines are to be changed, the convention is made that the change is substantive enough where a renumbering is desirable.

If the box has more than 100 lines, the syntax described above, which is XXX.YYY is changed to XXXYYYY, where the . is eliminated and the last Y is still left blank to handle enhancements.

Some examples are now in order. In these cases, we will show the progression of a number of lines of code from a black box to code:

For a black box, one would number the lines as follows:

```
B05.01    B05.01 if value = value of first S1 stimulus
B05.02    B05.02 then
B05.03    B05.03     R4:  Acknowledge stimulus;
```

For a state box, one would number the lines as follows:

```
B05.01    S05.02 if value = 3
B05.02    B05.03 then
          S05.04     value := S5;
B05.03    B05.05     R4:  Acknowledge stimulus;
```

For a clear box, one would number the lines as follows:

```
S05.02    S05.32 if value = 3
B05.03    B05.33 then
          C01.34     con
          C01.35         x, y := y, x;
```

For a clear box refinement, one would number the lines as follows:

```
S05.32      S05.98  if value = 3
B05.33      B05.99  then
C01.34      C01100      con
C01.35                  [x, y := y, x;]
            C02101          x := x + y;
            C02102          y := x - y;
            C02103          x := x - y;
```

If it was determined, after a verification, that the clear box refinement would need additional lines, one would number the lines as follows.

```
C01.34      C01100      con
C01.35                  [x, y, z := y, x, z + 1;]
            C02101      x := x + y;
            C02102      y := x - y;
            C02103      x := x - y;
            C02103a     z := z + 1;
```

Exhibit C: STATE DATA MODELING EXAMPLE

An example of state data modeling occurs when the Development Team is faced with a application that requires a departmental database for an organization. A database designer uses a modeling technique to represent the required semantics for the application. For the example, the Entity-Relationship model is used. The ER model is then translated into a relational database schema.

The problem statement for the database is:

> An organization has a number of departments, defined by a department number (D#), name (DNAME), a manager (MGR#), and a location (LOC). A department has employees, defined by employee number (E#), name (ENAME), position (POS), and salary (SAL). An employee works in one department. Each department is responsible for an equipment inventory. Each piece of equipment has attributes inventory number (INV#), description (DESC), and cost (COST). Employees work on projects, defined by project number (P#), project name (PNAME), and work site (SITE). An employee may work on many projects. The hours (HRS) worked on each project are recorded. Each project is administered by one department, and a department can administer many projects.

Entities are denoted by rectangles with attributes in attached ovals. A primary key is typically underlined. Relationships between entities are denoted by diamonds with arcs that connect the two entities in the relationship. Relationships are named and may have attributes in attached ovals. Three types of relationships may exist:

One-to-one relationship - An instance of one entity is related to at most one instance of the other entity and vice versa. For example, an employee can manage at most one department and a department is managed by at most one employee.

One-to-many relationship - An instance of the one entity is related to any number of instances of the many entity. However, an instance of the many entity is related to at most one instance of the one entity. For example, a department has many employees, but an employee is in at most one department.

Many-to-many relationship - An instance of an entity is related to any number of instances of the other entity and vice versa. For example, a project is worked on by many employees and an employee can work on many projects.

The cardinality of each relationship is shown with the characters 1 and M on the appropriate arcs.

Figure C-1 shows an Entity-Relationship model for the example database. There are four entities and five relationships in the database model.

The Entity-Relationship model can be transformed into a database schema. The following description demonstrates how a relationship database schema is developed from the ER model.

Each entity forms a separate relation. Relationships are represented by adding attributes to entity relations for one-to-one and one-to-many relationships or building new relations for many-to-many relationships. By matching the values of attributes among relations the relationships in the database are designed.

For the example database, each entity becomes a relation with the appropriate attributes:

DEPARTMENT (D#. DNAME, MGR#, LOC)
EMPLOYEE (E#, ENAME, POS, SAL)
EQUIPMENT (INV#, DESC, COST)
PROJECT (P#, PNAME, SITE)

The one-to-one manages relationship is already represented in DEPARTMENT by including the attribute MGR#. MGR# and E# in EMPLOYEE are defined on the same domain allowing values to matched across these two attributes.

One-to-many relationships are represented by placing a new attribute in the many entity that uniquely identifies the instance of the one entity to which it is related. For the works-in relationship between EMPLOYEE and DEPARTMENT, the unique attributed of DEPARTMENT, D#, would be added as an attributed to each employee, i.e.,

EMPLOYEE (E#, ENAME, POS, SAL, D#)

In the same way, the inventory and administers relationships are shown as:

EQUIPMENT (INV#, DESC, COST, D#)
PROJECT (P#, PNAME, SITE, D#)

A many-to-many relationship requires the construction of a new relation that contains an attributed for the primary key of both involved entities. Data that is recorded based upon the relationship of specific instances of each entity is stored as an attribute in this relation. Thus, the works-on relationship becomes the relation:

WORKS-ON (E#, P#, HRS)

Therefore, the final relational database schema for the example is:
DEPARTMENT (D#, DNAME, MGR#, LOC)
EMPLOYEE (E#, ENAME, POS, SAL, D#)
EQUIPMENT (INV#, DESC, COST, D#)
PROJECT (P#, PNAME, SITE, D#)
WORKS-ON (E#, P#, HRS)

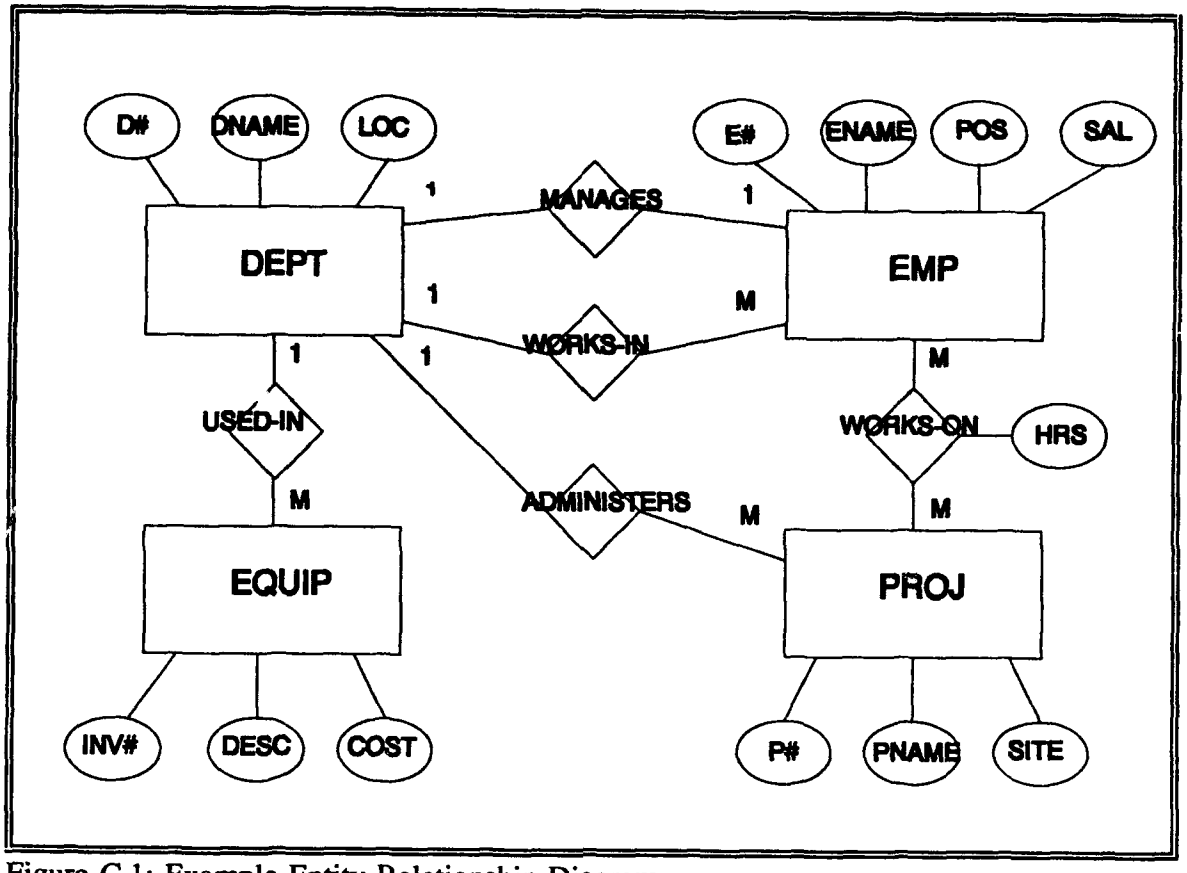


Figure C.1: Example Entity-Relationship Diagram